REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
06-09-99	FINAL	10-31-95 - 10-30-98

4. TITLE AND SUBTITLE
Towards HPC++: A Unified Approach to Parallel Programming in C++, Final Report under Contract Number DABT 63-95-C-0108

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Carl Kesselman

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
California Institute of Technology
CAL TECH 2136
1201 East California Boulevard
Pasadena, CA 91125-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Directorate of Contracting
Attn: ATZS-DKO-I
Post Office Box 12748
Fort Huachuco, AZ 8560-2748

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT
UNCLASSIFIED/UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Compositional C++ or CC++, is a general-purpose parallel programming language designed to support a wide range of parallel programming styles. By adding six new keywords to C++, CC++ enables programmer to express many different types of parallelism. CC++ is designed to be a natural extension to C++, appropriate for parallelizing the range of applications that one would write in C++. CC++ supports the integration of different parallel programming styles in a single application. It was designed to provide efficient execution on a range of parallel computing platforms, including both shared and distributed memory computers.

15. SUBJECT TERMS
Object Oriented Programming, Parallel Computing, Distributed Computing, Parallel Programming Languages

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPONE NUMBER *(Include area code)*

Towards HPC++: A Unified Approach to Parallel Programming in C++ Final Report for Contract Number: DABT63-95-C-0108

Principle Investigator
Carl Kesselman

1 Introduction

This Document is the final report for the Caltech subcontract Number PC 246143 under contract DABT63-95-C-0108, titled: Towards HPC++: A Unified Aproach to Parallel Programming in C++. In this report we describe the goals of the reasearch and summarize the major accomplishments.

1.1 Executive Summary

Compositional C++, or CC++, is a general purpose parallel programming language designed to support a wide range of parallel programming styles. By adding six new keywords to C++, CC++ enables a programmer to express many different types of parallelism. CC++ is designed to be a natural extension to C++, appropriate for parallelizing the range of applications that one would write in C++. C++ supports the integration of different parallel programming styles in a single application. It was designed to provide efficient execution on a range of parallel computing platforms, including both shared and distributed memory computers.

The main result of the project described in this report was to develop the technology required to implement CC++ and to produce a robust implementation of a CC++ compiler and the associated runtime systems needed to execute CC++ programs on a range of parallel and distributed computing environments, including workstations, shared memory and distributed memory parallel computers, and heterogenous networked collections of these machines.

In addition to developing CC++ compilation technology, this project has developed general tools that support the development of source to source transformations of programs. These tools have been used to support a range of other C++ research projects, for example at Indiana University, University of Oregon and Los Alamos National Laboratory.

One of the significant developments that occurred during this project was the emergence of Computational Grids as a next generation computing environment. The CC++ project has tracked this progress by basing its runtime system on the Globus Grid toolkit, enabling wide area execution of CC++ programs in a heterogenous environment.

1.2 Structure of the Rest of This Report

In the first half of this report, we focus on the CC++ language and how it is used to solve a range of parallel programming problems. We describe the design philosophy for the CC++language. We then demonstrate how CC++ can be used to solve a simple parallel programming problem: polygon overlay. In the second half of the report, we focus on implementation technology, describing the structure of the CC++ compilation environment, the design of SAGE-II, a source-to-source transformation tool for C++ programs and the design of a runtime library for CC++.

2 Design Philosophy and Goals of CC++

CC++ is the result of a desire to make research in compositional parallel systems available to a wider range users than were reached with previous compositional parallel programming systems such as PCN [CT91]. Briefly stated, a compositional parallel system is one in which all properties of program elements are preserved when those elements execute in parallel. We will discuss compositionality in more detail shortly.

Our previous work in compositional systems was based on designing new programming languages. While starting with a clean slate has its advantages, it throws up a significant barrier to getting many people to use the resulting system, no matter how clean and elegant. Based on this experience, CC++ adds essential elements of compositional programming systems into a widely-used language. C++ was chosen as a starting point because of its widespread use and its support for library construction, code re-use and programming in the large.

2.1 CC++ and Java

During the later stages of this project, the Java programming language has gained increased popularity. In many ways, Java is like a simplified C++, with

many of the more complex features, such as multiple inheritance and operator overloading, eliminated. In addition, Java provides an automatic garbage collection mechanism, which CC++ does not have.

While we considered the impact of Java on this project, we decided to continue our planned development path with CC++. Our reasons for this included:

- The performance of Java programs is still significantly poorer then the performance of C++ programs
- The Java Virtual machine does not run on all of the parallel architectures that were of interest to our research.
- Many libraries and applications of interest are implemented in C++, and not Java.

Finally, we note that while Java provides some limited support for parallel programming, including support for multithreaded programs, and a remote procedure call library (RMI), this support is quite limited and does not have many of the semantic and performance advantages of the extensions that make up CC++. We believe that many of the the results for this project can be directly applied to both the Java language and its implementation.

2.2 Compositionality and CC++

One of the reasons that writing parallel programs is harder than writing sequential programs is that it is difficult to reason about the behavior of a program in terms of the behavior of the pieces from which the program is constructed. A parallel programming system is said to be *compositional* if properties that hold for a part of a program in isolation still hold when that part of the program executes in parallel with any other piece of code.

To achieve compositionality, we must restrict the ways in which one component of a parallel composition can access data in another component. Consequently, parallel programming languages often limit the types of parallel computations that can be expressed. For example, one common approach is to limit a parallel programs to follow a data-flow computational model [BOCF92, Ack82, Gri93], requiring all shared variables to have the single assignment property, and a statement in the language cannot execute until all of its input variables have been given a value. In this model, all operations on data occur locally and atomically.

The advantage of these approaches is that they guarantee compositionality. However, there are drawbacks to enforcing a compositional computational model in the design of a parallel programming language:

- There are many different models for compositional programming. Restricting a program to just one of these can make implementation more complex if the application does not map well into the model being used.

- In some situations, non-compositional algorithms can yield better performance than that of their compositional counterparts.
- The pointer-based memory model of C++ can make it difficult to enforce the compositional model, or to support all of C++.

For these reasons, CC++ takes a less strict approach to compositional parallel programming. There are no restrictions on how a variable can be accessed within a parallel operation meaning that programs written in CC++, can have non-deterministic behavior. Rather than providing compositionality via a specific language-enforced mechanism, CC++ supports a wide range of compositional programming styles though language features and an approach to program design called the *proper interface* approach. The proper interface approach focuses on the properties that an interface between program components must have in order to ensure compositionality. The objective is to specify a software component clearly in terms of standard interfaces with clearly defined inputs and outputs. Features of CC++ such as interleaving semantics, explicit parallel blocks with known termination points, processor object encapsulation and synchronization variables facilitate the use of this approach. Specifics on the design of proper interfaces and the compositional approach to parallel programming can be found in [CK92]. [Siv94] and [Bin93] show how the compositional aspects of CC++ are used to prove properties of parallel programs.

2.2.1 CC++ extends C++

CC++ is a strict superset of C++. Any correct C++ program is also a correct CC++ program. However, this alone does not guarantee that CC++ "feels" like C++. One way we can ensure that CC++ makes sense to a C++ programmer is to apply the design philosophy of C++ to the design of CC++. Paraphrasing the designer of C++ [ES90], an important aspect of C++ design is to have the language consist of a set of basic features that can be combined to achieve solutions that would otherwise have required extra, separate features. C++ is designed to support a wide range of different programming styles, without forcing any particular programming style on the user.

In parallel programs, as in sequential programs, there are many different programming approaches, or paradigms. Some of the more common of these are data parallelism, task parallelism, object parallelism, and functional parallelism. Finer classifications are possible as well. For example, task parallelism can be built on shared memory programming primitives (such as locks, monitors and semaphores) or on message passing. In turn, message passing can be point-to-point or channel-based, and be either synchronous or asynchronous.

While it is convenient to speak of task or data parallel programs, there is no reason to restrict a program to a single programming paradigm. As the range of problems solved on parallel computers increases, it becomes desirable to construct *multi-paradigm* programs. In their simplest form, these programs combine

task and data parallelism. It has been demonstrated that such a combination can result in performance superior to single-paradigm programs [FBACX94]. We anticipate more general forms of multi-paradigm programs to prove important as well.

Consequently, CC++ does not provide specialized language constructs to implement a specific parallel programming model. Rather, it consists of a small number of constructs that when combined with C++ constructs allow us to implement a range of parallel programming paradigms. Parallel constructs in CC++ work in conjunction with the constructs in C++ to provide a basic set of mechanisms from which a wide range of paradigms can be constructed. Given an appropriate set of constructs, specific parallel programming paradigms can be implemented in the language as *paradigm libraries*.

The advantage of this approach is flexibility. Parallel programming paradigms can be specialized for the requirements of specific applications. Furthermore, by combining paradigm libraries, it is straightforward to integrate multiple programming paradigms within a single application. The overall result is that the complexity of parallel program development can be reduced.

2.3 Other Language Design Considerations

A desire to support compositional programming and parallel paradigm libraries, and to be consistent with the design philosophy of C++, all had an impact on the design of CC++. Some other concerns are:

Execution environment: CC++ programs should be able to exploit both shared and distributed memory parallel computers. The abstractions in CC++ should make it possible to port a program from one parallel architecture to another without recoding. On the other hand, CC++ also make it possible to exploit the characteristics of particular parallel computers without having to step outside the language.

Ease of learning: CC++ should be easy for a C++ programmer to learn. Our goal was to make it possible for a C++ programmer learn all of CC++ in half an hour. Learning CC++ is simplified by making most CC++ constructs explainable as generalizations of existing C++ concepts. Furthermore, all CC++ constructs have syntactic analogs in pure C++.

Migration path: CC++ should provide a migration path from an existing sequential application to a parallel application. This means that that all of C++ has to be included in CC++, including pointer operations and static and global variables. It also means that we must provide a way to encapsulate existing code in a parallel environment without change, and a way to introduce concurrency at arbitrary points in the program. Finally, it means that parallelism must not be tied to any specific language construct such as an object. Not all C++ programs are written in the

object-oriented style—if concurrency is tied to operations such as member function call, then the application must be rewritten in order to make it parallel.

Support for heterogeneity: We anticipate that parallel programs written in C++ will use several parallel programming styles. In addition, we expect parallel systems to be composed of different types of modules running on different types of machines. For example, an application might wish to run its visualization component on a shared-memory workstation while its computing component runs on a multicomputer. The abstractions provided by CC++ should support this type of heterogeneity.

Safety and efficiency: Finally, in designing CC++, it is important to keep on fact in mind: C++ is *not* perfect. The language has many safety features, but provides users with ways to sidestep them in order to make code more efficient or to simplify an implementation. For example, a pointer to a base class can be explicitly cast to a pointer to a derived class, even if the object being pointed to is not an instance of the derived class. Similarly, CC++ allows the explicit cast of a global pointer (discussed in Section 3.5.3) to a local pointer.

3 Summary of the CC++ Language

CC++ adds six new keywords to C++: the parallel composition operators `par`, `parfor`, and `spawn`, the synchronization constructs `atomic` and `sync`, and `global`, which is used to control of distribution and locality.

3.1 Parallel Composition Operators

CC++ programs are explicitly parallel. Parallel execution is specified by using parallel blocks (`par`), parallel loops (`parfor`), and spawned functions (`spawn`). The syntax of these statements is summarized in Program 3.1.

The execution semantics of parallel execution in CC++ is defined to be *fair interleaving*, which can informally be described in the following way. Consider the parallel block:

```
par {
  statement1;
  statement2;
  statement3;
}
```

in which each statement consists of a sequence of basic operations (i.e., assembly language instructions). Under fair interleaving, the operations of any one statement must execute in sequence. However, basic operations of two or more statements can be intermingled. Further, this interleaving must be such that

operations from each statement will execute eventually. For example, operations in *statement2* and *statement3* must eventually execute, even if *statement1* is:

```
while (1){
  foo();
}
```

```
// parallel block
par {
  statement 1;
  ⋮
  statement n;
}
```

```
// parallel loop
parfor (int i=0 ; i<n; i++)
  statement
```

```
// spawned function
spawn f(arg1, ···, argn);
```

Program 3.1: Parallel Control Structures in CC++

The most basic parallel construct is the parallel block or par block. Its syntax is modeled after that of the try block. The statements in the parallel block execute in parallel, using fair interleaving. The closing brace of the par block is a sequence point, as is the closing brace of a normal compound statement. This means that the statement after the par block does not execute until after all of the statements within the par block terminate.

The statements in a parallel block can be any C++ constructs except for variable declarations, gotos, or returns. Parallel blocks can contain normal compound blocks as well as nested parallel blocks. For example, the following is legal CC++:

```
par {
  {
    a();
    b();
    par {
      c();
      x = 1;
      y = x;
    }
  }
  par {
    g();
    h();
  }
```

```
  z = g(x+23);
}
```

There are no special rules on how variables can be accessed within a par block; all statement shares the same address space. This can obviously cause problems if care is not taken. In particular, the example above suffers from a race condition on the variable x. Other CC++ constructs must therefore be used to ensure that y has a sensible value.

A parallel block only specifies when statements can execute, not where they execute. In general, the statements in a parallel block, as well as the other parallel control structures in CC++, will execute at the same location. This implies that a new thread of control may have to be created to execute each statement in a par block. If the parallel block is running on a shared-memory computer, fair interleaving allows each statement to be executed on a different processor to achieve speedup. On distributed-memory computers, parallel control structures are combined with constructs that control locality to obtain speedups (Section 3.5).

Parallel iteration in CC++ is provided by the parfor statement. During execution of a parfor, each instance of the loop body is executed in parallel. The initialization, update and test parts of the parfor statement are evaluated sequentially, just as in a for statement, so that the execution of a parfor proceeds as follows:

1. Execute the initialization part of the statement
2. If the test is true, start the parallel execution of the loop body. If it is false, wait for all instances of the loop body to terminate.
3. Execute the update expression, then go to Step 2.

The semantics of the parfor statement is a result of the fact that C++ places no limitations on what can appear in the update, test and body of a for statement. Consequently, we can write:

```
parfor (list * ptr = head; ptr ; ptr = ptr->next)
  ptr->doit();
```

which iterates over a list and performs the operation doit() on each list element in parallel.

A consequence of the parallel execution of the loop body is that the value of the variables being used to control the iteration may have changed by the time the loop body corresponding to that iteration has a chance to execute. To resolve this problem, each loop iteration is provided with a local copy of all variables that are declared in the initialization part of the parfor statement. To avoid potential errors caused by changing the value of a loop index with the expectation that the value will propagate to another iteration, these copies are made constant:

```
parfor (int i=0; i<N ; i++){
  doit(i);        // reference local copy of i
  i++;            // compilation error
}
```

Because the other loop constructs in C++ do not allow for local variable declarations, CC++ includes only a parallel `for` loop.

`par` and `parfor` introduce parallelism in a structured manner: the statement after `par` or `parfor` does not execute until all of the statements executed by these constructs have terminated. The `spawn` statement provides an unstructured alternative: it causes the specified function to execute in parallel with the rest of the program and then terminates. Termination of the `spawn` statement does *not* imply completion of the function being spawned. The statement after a `spawn` statement executes without regard to the status of the spawned function.

The execution of the spawn statement is similar to that of a regular function call. First, the function expression and argument expressions are evaluated in an arbitrary order. The evaluation of the argument expressions is used to initialize the formal arguments of the function. At this point in the execution of a normal function call, the function body would start to execute. However, in the `spawn` statement, the next statement in the program starts executing as soon as the arguments to the function are copied. The execution of the function body is interleaved with that of the rest of the program. Any return value from the spawned function is discarded.

The unstructured nature of the `spawn` imposes some additional responsibilities on the programmer. Consider the following example:

```
{
  int x;
  spawn f(x);  // OK, x copied before spawn terminates
  spawn g(&x); // error: x can go out of scope
}
```

The spawned functions reference a variable that can go out of scope before either function has a chance to execute. This is not a problem in the case of `f(x)`, as the value of `x` is copied before it goes out of scope. The call to `g(&x)`, however, is dangerous, in that the pointer passed in as an argument does not necessarily refer to `x` when the function body executes. Note that an analogous problem can exist in a pure C++ when a pointer to a local variable is assigned to a dynamically-allocated pointer.

3.2 Synchronization Variables

Consider the following parallel code:

```
int x, y;
par {
  x = 1; y = x + 1;
```

}

Because of the interleaving that occurs in the parallel block, there have no way of knowing what the value of y will be. We therefore need some way to control how the operations in a CC++ program are interleaved. Synchronization, or sync, variables are one of two mechanisms provided in CC++ for this purpose, atomic functions being the other.

A sync variable is a single-assignment variable [Ack82]: it can be assigned a value at most once, and its value cannot be used until the assignment has taken place. Another way to view this is that a sync variable is just like a const variable with delayed initialization. Any attempt to use the value of a sync variable prior to initialization will cause the reading thread to block until some time after the variable has been initialized. Once the variable has been initialized, it behaves just as if it had been declared const. In fact the syntax for declaring a sync variable is identical to that of a const variable.

Several examples of sync variable declarations are shown in Program 3.2. The first line in this example shows the normal C++ declaration for a constant integer. The declaration of a sync variable on the next line is similar. A sync variable can exist for any type that can be declared const; for example, the third declaration is for a sync pointer to an integer. In this case, we can synchronize on the existence of the pointer, but not on the object being pointed to.

The use of a sync variable is shown in the function f(). Note that the argument to f() is a reference to a sync integer. Since we are not asking for the value of the sync integer, calling the function f() will not block. Looking at the function body, the parallel block it contains allows the statements to execute in any order. However, because of the sync declarations, the statement that sets the value of sync_x must execute before the statement that uses sync_x. Because the third statement in the parallel block attempts to update the value of a sync variable, it is an error.[1] While this particular error can be detected at compile time, in the general case, such errors cannot be detected until runtime.

```
const int x = 23; // constant integer
sync int sync_x; // sync integer
int * sync iPtr; // sync pointer to an integer

void f(sync int & sync_int){
  int y, z;
  par {
    // waits for assignment to sync int
    y = sync_int + 1;
    // initialize the value of the sync variable
    sync_int = 23;
    // compilation error
    z = sync_x ++;
```

[1]The ++ operator must read before it can write.

```
  }
}
```

Program 3.2: Examples of sync Variables

3.3 Atomic Functions

Sometimes it is necessary to prevent certain interleavings from occurring within a parallel execution. In CC++, this can be achieved by performing the actions within the body of an `atomic` function. An atomic function is declared by adding the specifier `atomic` to the function's declaration:

```
class Queue {
  atomic void insert_into_queue(T);
}
```

Within an instance of a C++ class, only one `atomic` function may execute at a time. Regular C-style functions and static member functions can be declared `atomic` as well, providing atomicity within an instance of a processor object (Section 3.5).

The execution of an `atomic` function may be interleaved with the execution of `atomic` functions in other classes, with `atomic` functions in other instances of the same class or with non-atomic functions in the same instance of the class. In addition, an `atomic` function can directly call another `atomic` function within the same object without blocking.

An alternative to `atomic` functions would be to sequentialize access to an object and only allow a one member function at a time to execute within an instance of a class. This method has the advantage that one does not have to be concerned with the potential interactions between member functions. One reason we choose not to take this approach is that it limits ways in which C++ classes can be used in parallel applications. Consider the following example:

```
class profiled_foo {
  foo_member() { counter++; big_function(); }
  const counter_value() { return counter; }
private:
  int counter;
};
```

In this situation, there is no reason why access to the counter should be restricted while `big_function` is executing. If we had required sequential access to `profiled_foo`, we could not have written this type of class. Since a user can sequentialize access by making all of the public member functions of a class atomic, CC++ does not require sequentialization as part of the language semantics.

We note that `atomic` functions have the identical semantics to the Java `synchronized` member functions. However, unlike `atomic` functions, which

manage threads created by control structures in the language (i.e. par and parfor statements), Java synchronized functions control threads created by a separate Thread class library.

3.4 Implementing a Blocking Lock

On of the features of CC++ is that its primitives can be used to implement a range of different types of parallel programming paradigms. In this section, we show how sync variables and atomic functions can be combined to implement a parallel programming construct that is not part of CC++: a blocking lock. The lock is implemented as a CC++ class, and used as in:

```
Lock lock;
int i;
par {
  { lock.lock() ; i++ ; lock.unlock(); }
  { lock.lock() ; i++ ; lock.unlock(); }
}
```

The implementation of the lock contains a variable that records the lock's state. CC++ atomic functions are used to ensure that operations on a Lock leave it in a consistent state. Because an attempt to obtain a lock that is already held must suspend, the function that implements the lock request cannot be made atomic. Otherwise, it would prevent the required unlock operation from taking place, resulting in deadlock. Therefore, the function implementing the lock request uses an auxiliary function, which is made atomic. The unlock operation, however, must atomically test the state of the lock and either awaken a blocked thread or reset the state of the lock to unlocked.

The declaration for this Lock class is given in Program 3.3. Its interface declares three functions. The use of the functions lock() and unlock() should be clear. As discussed above, only unlock() is declared atomic. The class's constructor ensures that locks are always initially unlocked.

```
class Lock {
 private :
  enum lock_state {LOCKED, UNLOCKED};
  lock_state state;
  queue<sync int *> waitingQueue;
  atomic check_lock();
 public :
  void lock();
  atomic void unlock();
  Lock() { state = UNLOCKED; };
}
```

Program 3.3: Interface of CC++ Lock Class

The private part of the lock contains its implementation. It first declares a private type, which is used to indicate the state of the lock, and a variable of this type to hold its state. The variable waitingQueue is a queue of pointers to sync integers, implementing using the queue class from the Standard Template Library. Finally, we declare that a Lock has an atomic function called check_lock(), which is the auxiliary atomic function used by lock().

The member functions of Lock are given in Program 3.4. The lock() method begins by allocating the sync variable got_lock. This variable is used to indicate when the lock is granted to this lock request. Note that got_lock is allocated on the stack, so that its storage is automatically reclaimed when it goes out of scope at the termination of a lock() call. For this memory allocation strategy to work, we need to ensure that there are no outstanding references to this variable once the lock is granted—which is in fact the case in the Lock class. After allocating got_lock, the atomic function check_lock() is then called with a pointer to got_lock as its argument. On returning from check_lock(), we examine the contents of got_lock. This blocks execution until got_lock has been initialized, which indicates that this call to lock() has been granted the lock.

The function check_lock() tests to see if the lock is already held. If so, the pointer to the sync variable is pushed onto the waiting queue and check_lock() returns. Otherwise, the state of the lock is set to LOCKED, the sync variable is initialized, and check_lock() returns. Because check_lock() is an atomic function, CC++ ensures that interleaving will not change the state of the lock between the time the state is checked and the time the pointer to got_lock is enqueued.

```
void Lock::lock(){
  sync int got_lock;
  check_lock(&got_lock);
  // check if got lock is initialized
  got_lock == 1;
}

atomic void Lock::check_lock(sync int * go){
  if (state == LOCKED)
    waitingQueue.push(go);
  else
    *go = locked = LOCKED;
}

atomic Lock::unlock(){
  if (waitingQueue.empty())
    state = UNLOCKED;
  else
    // Grant lock to waiting request
    *waitingQueue.pop() = 1;
```

}

Program 3.4: Implementation of Member Functions of `Lock` Class.

To release the lock, we must first check the waiting queue to see if any threads are blocked. If so, we dequeue the pointer to the appropriate `got_lock` and initialize it. This allows a blocked `lock()` call to proceed. If there are no threads waiting for the lock, we simply change the state of the lock to `UNLOCKED`.

3.5 Specifying Location in CC++

So far, all of the CC++ constructs introduced deal with specifying when operations can take place. The remaining CC++ constructs—processor objects and global pointers—are used to specify *where* operations can execute. In addition to providing a mechanism for specifying locality, these constructs address a number of additional design issues, including:

- a means of abstracting processing resources in the programming language;
- a mechanism for separating algorithmic concerns from resource allocation issues;
- a means for describing heterogeneous computation; and
- a mechanism by which existing C++ codes can be properly composed from within a parallel block.

All of these issues are addressed by the CC++ concept of a *processor object*. In C++, a computation consists of a single instance of a program, which is constructed by linking together one or more translation units, or files. In CC++, we generalize the idea of a computation to include multiple instances of more than one program: that is, a computation can have many instances of the same program, or many instances of different programs. We call each instance of a program in a CC++ computation a *processor object*.

A processor object is defined by linking one or more translation units with a *global class* declaration. Associating a class declaration with a program enables CC++ to treat a C++ program as a regular C++ object. A processor object has a type, and can contain member functions, data members, nested class and type definitions, constructors, destructors, and so on. A processor object can by dynamically created with the `new` operator and destroyed with the `delete` operator. Program 3.5 shows that a global class definition looks like a regular class definition, except for the addition of the keyword `global`. As this file defines two different global classes, it in effect declares two different types of programs.

// file: global defs.H

```
global class ProgramA {
  f();
}

global class ProgramB {
 public :
  f();
 private :
  g();
};
```

Program 3.5: Examples of Global Class Declarations

As with any class declaration, the global class declaration specifies the data and function members of the class. The translation units or files that are linked together can contain the implementation of these members. Processor object definitions are created by the CC++ compiler at link time. For example, consider the global class declarations in Program 3.5 and the source code in the file listed in Program 3.6. Using the CC++ compiler developed under this project, we type:

```
cc++ -o program_a -ptype ProgramA file1.cc++
```

to create a processor object of type `ProgramA`. The implementation of this processor object is placed in the file `program_a`.

By executing a file containing the implementation of a processor object, we can start a CC++ computation. Initially this computation contains a single processor object whose type is that specified by the `ptype` argument to the compiler. In our example, executing `program_a` would start a CC++ computation that contained one processor object of type `ProgramA`.

The first processor object in a computation is a special case. Once it has been created, the member function

```
int main(int, char**)
```

is run. Only the first processor object in a computation is required to have a `main` function and the `main` function is only run in the initial processor object in a computation.

```
// Contents of file1.cc++
#include "global_defs.H"
int main(int, char **) { ··· }

ProgramA::f() { ··· }
```

Program 3.6: Source code in file1.cc++.

Lets consider a more complicated example, shown in Program 3.7. We can create a processor object definition from this file with the compiler command:

```
cc++ -o program_b -ptype ProgramB file2.cc++
```

Unlike the previous example, `file2.cc++` contains a class declaration, `A` and a global variable, `x`. These declarations receive special treatment when the processor object definition is put together. The obvious thing to do is to simply include these declarations as members of the global class. However, this would make it difficult to integrate existing code and libraries into a CC++ program. Even if one had access to all of the variables and classes in a existing program, requiring a user to manually enter all of the declarations into the global class declaration would be awkward.

```
// Contents of file2.cc++
#include "global_defs.H"
class A {};

ProgramB::f() { ··· }
ProgramB::g() { h() }

int x;

h() { ··· }
```

Program 3.7: Source code in file2.cc++.

Our solution to this problem is to consider all top-level declarations in a translation unit to be implicitly included as private members of a processor object. Thus in Program 3.7, the class `A` is a private, nested class in a processor object of type `ProgramB`, while the variable `x` is a private data member of the processor object.

One remaining issue is what to do about objects with external linkage. For example, consider the following class declaration:

```
class C {
static int data_member;
}
```

Normal C++ semantics say that the name `data_member` refers to the same piece of storage in every instance of class `C`. Extending these semantics to processor objects would result in sharing memory between processor objects, which would be contrary to our desire to use the processor object to specify locality. Accordingly, we modify the linkage rules of C++ to state that externally-linked names are resolved to the same object within an instance of a processor object, but are treated as if there were internally linked between instances of a processor object.

3.5.1 Creating New Processor Objects

As we saw above, one way to create a processor object is to execute a file containing the implementation of that processor object. Additional processor objects can be created through the use of the new operator.

If we have a global class definition:

```
global class worker {
 public :
  worker();
  worker(int worker_id);
  int do_work();
  int status;
}
```

then the expression:

```
worker * global gPtr = new worker;
```

will create a new processor object of type worker and return a pointer to it. (We explain the use of the keyword global in Section 3.5.3). In this example, the default constructor for the processor object will be called. Note that creating a new processor object does not create a new thread of control.

As with any other CC++ object, members of a processor object can be accessed though a pointer to that object. Upon executing the variable declaration shown above, the statements:

```
wPtr->status++;
wPtr->do_work();
```

will increment the value of status in the newly created worker processor object and call the do_work function.

We can combine C++ library-building constructs with processor objects and parallel control structures to achieve speedup on a distributed memory machine. Given the worker class above, we want to create a collection of workers, and then call the do_work() function on each worker in parallel. We would also like to hide the details of processor objects from the end user. An implementation of this program is found in Program 3.8.

```
class worker_ptr {
 public :
  static init_workers(int n) {
    workers = n;
    for (int i=0; i<n; i++) worker_array[i] = new worker;
  }
  worker global * operator->(){
    return worker_array[next_worker = (next_worker + 1) % workers];
 private :
```

```
  static int next_worker;
  static worker_array[MAX_WORKERS];
};

use_workers() {
  worker_ptr::init_workers(128);
  worker_ptr ptr;
  parfor (int i=0; i<1024; i ++)
    ptr->do_it();
}
```

Program 3.8: Using Processor Objects to Achieve Parallelism

The parallel computation is performed in the function `use_workers`. Before we can do any work, we must create the processor objects on which the computation will take place by calling the static member function:

```
worker_ptr::init_worker()
```

This function creates processor objects of type `worker` and stores pointers to them in the static private data member `worker_array`. In practice, we will want each processor object to be located on a different physical processor; we show how this can be achieved below.

With the processor objects created, we are ready to perform a parallel computation using the function `use_workers()`. After performing the initialization, this function creates the variable `ptr`, which is of type `worker_ptr`. Within the body of a `parfor` loop, we call the `do_it()` function using `var` as a pointer. This expression calls the overloaded `->` operator in `worker_ptr`, which returns a pointer to a processor object. The exact processor object used depends on the number of workers and the number of times the `->` operator has been called. Thus the class `worker_ptr` not only isolates `use_workers()` from the details of processor objects but also provides a means to separate the mapping of computation onto processor objects from the body of the function as well. Using the global pointer returned by the overloaded `->` call, the function `do_it()` is then called.

This example demonstrates a number of interesting aspects of CC++. It shows how "where" constructs can be combined with "when" constructs to create a parallel program. It also shows how CC++ constructs can be combined with C++ constructs to encapsulate the mapping of computations to resources. While computation was distributed to processor objects in a round-robin manner in this example, more complex algorithms could be easily encapsulated in the `worker_ptr` class.

3.5.2 Specifying the Placement of a Processor Object

As we have seen, processor objects provide a convenient abstraction for talking about the distribution of computation in a CC++ program. We now discuss the

method by which we can control the mapping of processor object onto physical computing resources.

The C++ `new` operator actually calls a function `new(size_t)`, where `size_t` is an implementation-defined type. Additional arguments can be passed to an overloaded `new` by using the so-called placement syntax. For example, the expression:

```
new (23) T
```

calls the function `new(sizeof(T),23)`. Placement is typically used to force the C++ runtime system to use a particular region of memory when allocating an object.

When the type of the object being allocated is defined by a global class, a function whose signature is `new(proc_t)` is called. `proc_t` is a regular C++ type defined by an implementation of CC++, just like `size_t`. However, where a `size_t` specifies the size of an object, a `proc_t` specifies the location of an object. If no placement argument is specified, the `new` function is called with the same `proc_t` that was used to create the processor object on which the `new` expression is being evaluated. Otherwise, the `proc_t` provided via the placement argument is passed as the first argument to the `new` function.

Returning to class `worker`, we can specify that the allocated processor objects be located on node 34 of a parallel computer named `bigboy` with the following code:

```
// Include definition of proc_t
#include <stddef.h>

// Create a proc_t that specifies node 34 on bigboy
proc_t where("bigboy#34");

// Create an instance of worker on a specific node
// Pass an argument of 10 to the constructor
worker * global gPtr = new (where) worker(10);
```

3.5.3 Global Pointers

In Section 3.5.1, we saw that the type of the pointer returned from a processor object allocation had the type specifier `global`. In CC++, a pointer that is used to reference between processor objects must have this type specifier in its declaration. We refer to such a pointer as a *global pointer*. Declarations of global pointers look just like declarations of constant pointers. For example:

```
// global pointer to an int
int * global p1;

// global pointer to a sync int
sync int * global p1;
```

```
// constant global pointer to a pointer to a sync int
sync int * * const global p1;
```

Global pointers give CC++ an explicit two-level locality model: objects are either close and inexpensive to reference, or (potentially) far away and expensive to reference. Note that just because a pointer is global, it does not mean that it will be costly to get to the data it references, just that it may be.

We can have a global pointer to any C++ object. There is an implicit conversion from a local pointer to a type to a global pointer to that type. Thus:

```
int x;
int * global iPtr = &x;
```

produces a global pointer to the variable x.

3.5.4 Moving Data Between Processor Objects

The final aspect of CC++ to be explained is how data is moved between processor objects. This happens whenever an operation takes place through a global pointer. For example, the following statements all have to copy data from one processor object to another:

```
class A;
class C {
  // A function with a class as its argument and return value
  C foo(A);
};

A a;
C * global cPtr;
int * global gPtr;
int x;

copy an integer between processor objects
x = *gPtr;
*gPtr = x;

// Copy instance of classes A and C between processor objects
C retvalue = cPtr->foo(a);
```

Transferring data from one processor object to another is similar to the problem of generating copy constructors for a data type. The solution used to generate copy constructor is to perform member-wise copies, where built-in types are copied bitwise. This method generate a sensible copy operation as long as the data type being copied does not contain a pointer.

The CC++ compiler does the same thing to move data between processor objects. To move data between processor objects, the data must first be packed into an architecture neutral format. Conceptually, the packed data does not live

on any processor, but rather in a "space" that exists between processor objects; we call this space the *void*. Data transfer consists of inserting data into the void from the source processor object and extracting data out of the void on the destination processor object. To enable this process, the compiler generates a pair of functions: an into-the-void function and a out-of-the-void function. The compiler automatically generates `void` functions by calling the appropriate `void` function on the members of the data structure.

In the situation where this is not the desired behavior, the user can specify a `void` function. For a type `T`, the `void` functions have the following signatures:

```
global void & operator << (global void & v, const T & data);
global void & operator >> (global void & v, T & data);
```

Typically, one must define a pair of void functions in the same circumstances where one has to define a copy constructor and assignment operator.

4 A Programming Example: Polygon Overlay

To demonstrate the use of CC++ to support different parallel programming examples, we consider a complete sample program. Two different parallel implementations will be given. The first is designed specifically for shared memory parallel computers. From this version, we will create a version of the program that can execute on distributed memory parallel computers.

The example that we consider is the polygon overlay problem, in which one computes the geometric intersection of two input maps each of which defines an alternative decomposition of a geometric space into non-overlapping polygons. The basic algorithm for computing the overlay for each polygon in the first map, compare its coordinates with every polygon in the second map, comparing the coordinates of the two polygons and creating a new polygon representing the overlap, if any. Optimizations of the basic algorithm include sorting the second polygon list to reduce the number of elements that must be checked and keeping track of the overlap area in the polygons of the second list so that a polygon can be permanently taken out of consideration if it is completely covered.

We started with an existing implementation of polygon overlay, written in ANSI C. Our approach to producing a parallel program is to make incremental changes, preserving as much of the original program as possible. The first step in this process is to ensure that the existing C code can be parsed as C++. Since the program is ANSI C, and since CC++ is a superset of C++, we were able to compile the sequential polygon overlay code with the CC++ compiler without change. With this done, we can modify the program to introduce parallelism.

4.1 A Shared Memory Implementation of Polygon Overlay

The main computational kernel of the polygon overlay algorithm is shown in Program 4.1. The doubly nested loop checks every element in the first (left) map against every element in the second (right) map. Overlay regions are stored in a linked list, whose head is pointed to by the variable `outList`.

```
// pl and pr point to elements in the left and right map
for (il=0, pl=left->vec; il<left->len; il++, pl++) {
    for (ir=0, pr=right->vec; ir<right->len; ir++, pr++) {
      // polyOverlayLn allocates new list cell when overlap exists
      if ((newLn = polyOverlayLn(pl, pr)) != NULL) {
        newLn->link = outList;
        outList = newLn;
      }
    }
  }
```

Program 4.1: Kernel Operations in polygon overlay application

An obvious way to construct a parallel version of this application would be to replace the two `for` loops with CC++ `parfor` loops, modifying the body of the inner loop to ensure that concurrent updates to the output list are handled correctly. However, there is a performance problem with this solution that must be considered. Ideally one would like to use a parallel block or loop structure whenever parallel execution is possible. However, in practice, there is a cost associated with creating a new thread of control on most current computer architectures [2]. Consequently, simply replacing the `for` statements with `parfor` statements will result in an unacceptable level of overhead.

The solution to this problem is to introduce a limited amount of parallelism by adding a single `parfor` loop around the existing sequential loops. The parallel loop slices the first map into a fixed number of sections, where calculations on the sections can execute in parallel. Within a section, the `for` loops compute the overlays sequentially.

We now turn to the problem of updating the output list. There are two basic methods by which concurrent updates to the output list can be made. One approach is to introduce a list class that supports atomic operations. However, requiring atomic update for every list modification can be costly. Consequently, our approach is to batch updates into a local output list and the to atomically append the local lists together to form the overall solution. Here is an implementation of the append operation:

```
// Atomically append list l2 to list l1.
```

[2] There are a class of multi-threaded computer architectures in which thread creation is inexpensive, however, these computers are not yet widely used

```
atomic atomic_append(PolyLn_p l1, PolyLn_p l2) {
    PolyLn_p end_ptr = l1;
    // Find the end of the first list.
    while(end_ptr->link != NULL) end_ptr = end_ptr->link;
    end_ptr->link = l2;
}
```

Putting everything together, Program 4.2 shows all of the changes required for a shared memory implementation of the polygon overlay application. The variable `parallelism` controls how many overlay computations will be taking place concurrently. Each iteration of the `parfor` statement creates a local output list which it then appends into the list, `outList` to construct the final solution.

```
// Compute overlay on left map slices in parallel
parfor (n = 0 ; n < parallelism ; n++) {
   // List to store local overlay computation
   PolyLn_p local_outlist = NULL;
   for (il=n, pl=left->vec;
        il<left->len;
        il += parallelism, pl += parallelism) {
          for (ir=0, pr=right->vec; ir<right->len; ir++, pr++) {
            if ((newLn = polyOverlayLn(pl, pr)) != NULL) {
              newLn->link = local_outList;
              local_outList = newLn;
            }
          }
   }
   // Add local overlays to global list
   atomic_append(outList,local_outList);
}
```

Program 4.2: Shared memory kernel for polygon overlay application

4.2 A Distributed Memory Implementation of Polygon Overlay

In the shared memory implementation of polygon overlay using CC++, we created multiple threads of control in a single processor object. To construct version of this program that can execute on distributed memory computers, we must create more than one processor object. In the shared memory polygon overlay program, parallelism is introduced by slicing the first map into sections and processing the sections in parallel. The same approach is used to introduce parallelism in the distributed memory version as well. However, in the shared memory code, the sections of the first map are implicitly defined by the `parfor`

loop. In the distributed memory version, the slicing is explicit as each section must be placed into a different processor object.

Notice that using more than one processor object does not prevent us from having more than one thread of control in the processor object. The shared memory and distributed memory versions of the polygon overlay code can be easily combined into a single application that exploits both types of parallelism. This would be appropriate for execution environments that support both shared and distributed memory programming models. Examples include parallel computers with multi-processor nodes, such as the Intel Paragon MP, or networks of multiprocessor workstations.

The top level of the distributed polygon overlay code is shown in Program 4.3. To simplify the development of this code, we use components from the standard CC++ library. This library is written completely in CC++ and contains reusable components that implement a variety useful parallel programming paradigms and abstractions. The declaration of class `polyoverlay_array` uses the CC++ library to define a *processor object array*, a homogeneous collection of processor objects. Each element of the `polyoverlay_array` contains an reference to an instance of class `polyOver` as well as references to the other elements of the array. Another class defined in the library is the `locations` class, which is just an array of `proc_t` objects. A `locations` object can be used when a processor object array is created to specify where the elements of a processor object array are over nodes of a computer.

```
class polyOver {
public:
    // Interface to sequential polygon overlay computation
    polyVec_p doOverlay(polyVec_p left,polyVec_p right);
};

// Declare a array of processor objects whose
// elements are of type polyOver
global class polyoverlay_array {
    // Make this class a processor object array
    declare2(pobj_array,polyoverlay_array,polyOver)
public:
    // Initialize processor object array
    polyobj_array_test(locations locs, const int sz) :
        array_components(locs, sz,this) {}
    polyVec_p doOverlay(polyVec_p, polyVec_p);
};

int main(int argc,char **argv)
{
    polyVec_p leftVec, rightVec;

    int nodes = atoi(argv[1]);
```

```
    // Read data from map files.
    leftVec = polyVecRdFmt(argv[2]);
    rightVec = polyVecRdFmt(argv[3]);

    // Create processor objects
    polyoverlay_array array(nodes);
    polyVec_t outVec = polyover_array.doOverlay(leftVec, rightVec);

    // This sort could be parallelized as well
    qsort((void *)outVec.vec, outVec.len, sizeof(poly_t), polyCmp);

    // Write out the result
    polyVecWrFmt(argv[4], &resultpolyVec);
    return 0;
}
```

Program 4.3: Top level code for distributed memory implementation of polygon overlay application

When the distributed polygon overlay application is started, the number of nodes to use, the file names for the input maps and the output file are passed as arguments. The data from the map files is read and the contents stored into the vectors: `leftVec` and `rightVec`. We then create and instance of the processor object array `polyoverlay_array`, which creates `nodes` processor objects. We call the member function `doOverlay`, which returns the a polygon vector containing the results. After sorting the results, we can output them to a file.

The member function `polyoverlay_array::doOverlay` is responsible for performing the parallel computation. Conceptually its implementation is quite simple. It iterates in parallel over the elements of the array, calling the function `polyOver::doOverlay`. This function returns the the overlay of the polygon vectors passed in as arguments as a polygon vector. The first argument to `polyOver::doOverlay` contains the slice of the first map to be processed by the processor object, while the second argument contains the entire second map. The calls `polyOver::doOverlay` are made through global pointers created by the constructor for the processor object array. Thus into and out of the void functions are used to transfer the polygon vectors between processor objects. While these must be defined as part of the parallel implementation of the parallel polygon overlay code, they are easy to construct as the polygon vector consists of a vector length and a vector of integers.

The implementation `polyOver::doOverlay` contains the computational kernel from Program 4.1. As we mentioned previously, we could just as well use the version from Program 4.2, allowing our code to exploit both shared and distributed memory. With the exception of one minor modification, we use the sequential polygon overlay implementation unchanged. The alteration to the basic

polygon overlay algorithm improves the performance of `polyOver::doOverlay` when processing the first map in slices. The input polygon vectors are sorted by the x coordinate of the left-hand side of the polygons in the vector. Since the first argument to each instantiation of `polyOver::doOverlay` only has a slice of the entire map, it does not cover entire region. If we determine the range of the x values covered by the vector slice of the first map, an initial portion of the second map can immediately be eliminated from consideration in overlap computations. This optimization is implemented by `polyOver::doOverlay`.

Performance results from the distributed memory version of the polygon overlay code shown in Table 1. These runs were made on an IBM SP-2. Each node has 128 Mbytes of memory.

Node	Sequential	2	4	8	16	32
Time						
Speedup	1	2	4	8	16	32

Table 1: Performance results of distributed polygon overlay code on an IBM SP-2

5 Implementing CC++

Much of the work completed during this project was to produce an robust CC++ compiler, capable of enabling application development. The structure of the compilation environment is shown in Figure 1. The CC++ compiler is implemented via a source to source transformation. CC++ code is parsed and transformed into ANSI C++, with calls to a CC++ specific runtime library. The resulting code is then compiled by the vendor supplied C++ compiler and linked with the CC++ runtime library to generate an executable. A top level driver script hides the details of this process from the end user, allowing them to invoke the CC++ compiler just like any other compiler on the target system.

Structuring the CC++ compiler as a CC++ to C++ transformation has several advantages. First, it simplifies the process of porting and maintaining the compiler, as architectural specific dependencies are minimized. In addition, the use of the vendor supplied C++ compiler for code generation can simplify porting existing applications to CC++. Within the C++ standard, there are a number of behaviors that are implementation specific. For example, example, the lifetime of temporary variables generated by the compiler can vary from implementation to implementation. While it is not good programming style to depend on these implementation dependencies, it is desirable that a working C++ program continues to function correctly when compiled by the CC++compiler. By relying on the vendor compiler for code generation, we can

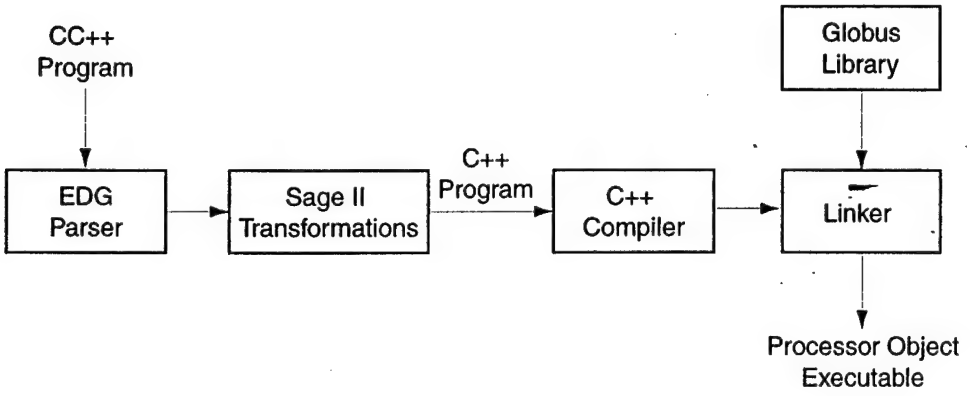

Figure 1: Structure of the Compositional C++ compiler

ensure that the behavior of existing C++ code will be unchanged when passed through the CC++ compiler.

One disadvantage of using source to source compilation to implement the CC++ compiler is that there are some optimizations that can be performed when generating object code, but cannot be expressed in the generated C++ code. For example, the semantics of thread creation, variable sharing and thread synchronization could be exploited during the code generation stage to optimize memory allocation and register usage. However, we believe that these limitations of our approach are far outweighed by the advantages discussed above.

In the following sections, we will discuss the details of the compiler as implemented under this contract. This will include the parser, a source to source transformation tool that we developed, the compiler transformations, and the CC++ runtime library.

5.1 Parsing CC++

The goal of the parsing stage of the compiler is to process the textual representation of a program an to produce a set of data structures that can be manipulated by later stages of the compiler. ANSI C++, and hence CC++, is a large and complicated language. Tokenizing is complex and often context dependent and the language is best parsed by a hand crafted recursive decent parser, rather then by automatically generated parsers such as those produced by tools like YACC. The situation is further complicated by the need to handle the instantiation of C++ templates, which require handling of partial type information and introduce the need for link time instantiation of missing templates. Because

CC++ extensions were designed to be minimal and consistent with the design of the rest of C++, our approach to parsing was to extend an existing C++ parser to accept CC++ syntax, rather then create a CC++ parser from scratch, given the complexities described above.

We selected a C++ parser developed by the Edison Design Group (EDG) the basis of our CC++ parser. We made this choice for several reasons:

- It was a high-quality implementation and served as the front end for many commercial C++ compilers.
- The parser was actively tracking the developments in the ANSI C++ standardization committee,
- academic institutions could get access to source code for no fee, and we could freely redistribute binary copies of the parser.

The parser consists of over 250,000 lines of C source code. We made modifications to the tokenizer, parsing routines, and symbol table routines, among others, to support the the CC++ language extensions. We also modified the template mechanisms so that CC++ extensions were smoothly integrated into the process of template instantiation. In performing these modifications, particular attention was paid to ensuring the accurate and meaningful error messages were produced by the parser when parsing CC++ code.

As provided by EDG, the compiler could be configured with a range of back-end processes. We augmented the compiler with a back end of our own design to produce a Sage V2.0 intermediate form (described below) and to perform the CC++ specific transformations that implement the actual compilation process.

5.2 Representation of Parsed C++ and CC++ Programs

As provided by EDG, the parser represents a C++(or CC++) program file as a collection of symbol tables and a set of C data structures. While the EDG developed representation provided a complete representation of the parsed program, its format was complex and difficult to manipulate. As part of our goals for this contract, we wanted to develop general compilation infrastructure for the C++ research community. Consequently, we undertook to develop a program representation that was easier to use and specifically designed to support the development of source to source transformations. The results was a tool called Sage V2.0, which provides an object-oriented representation of a C++ program and a set of operations that facilitate the expression of program transformations. Sage V2.0 was developed in collaboration Indiana University, as part of their DARPA funded HPC++ project.

Sage V2.0 is implemented as a C++ class library. Objects in this library represent program object such as statements, variables, expressions, etc. In order to simplify the process of designing transformations, Sage representations

were designed to be as high level as possible, ideally a direct encoding of the literal program structure. We note that there are situations where this is not possible, as such a representation will hide important details about the structure of the program. However, in general, we were able to design representations that were much simpler than those used internally by the EDG parser.

To support source to source transformation, the Sage V2.0 library provides a set of iterators for looping over program functions, statements and elements of expressions. Use of these iterators is simplified by structuring them to look like the iterators defined in the ANSI standard template library. Each class also provides an unparse function, which is used to convert the Sage representation into an text file, suitable for input to a backend C++ compiler.

Program 5.1 illustrates the use of Sage 2.0 in a simple transformation found in the CC++ compiler: reading the value of a variable declared to be of type sync. As we will discuss below, the CC++ compiler iterates over the statements and expressions in a CC++ program. During this iteration, it looks for variable with the type modifier sync and converts them to be instances of a corresponding C++ class. For example, sync int variables are converted to be of type SyncInt. This class has a member function called value which blocks until the variable has been assigned a value, and returns that value once it exists. The sync variable transformation therefore is to transform a code such as:

```
sync int x;
int y;
y = x + 23;
```

into

```
class SyncInt; // defined in cc++ runtime library

SyncInt x;
int y;
// this will block until x is set
y = x.value() + 23;
```

To perform this transformation, the CC++ compiler iterates over all of the expressions in a program looking for a reference to a variable with the sync type modifier. For each such variable, the compiler will call the function addDotValue, shown in Program 5.1. This function looks up the declaration of the .value member function in the SyncInt class and creates a new expression to replace the variable reference.

```
// Transformation for reading a cc++ sync variable.  To
// ensure correct synchronization each variable is represented
// as an class, with a value() member function that blocks until
// the variable has been assigned a value.

SgExpression *applyCheckSync::addDotValue(
```

```
    SgType* n,
    SgType *t,
    SgExpression *e
)

  SgExpression *new_expr;

  // Underlying type is sync, but may be a references, so
  // skip over the reference part of variable if this is the case
  SgType *tmp_n=n;
  if(tmp_n->variant()==T_MODIFIER)
    tmp_n=((SgModifierType *)tmp_n)->get_base_type();

  SgClassDeclaration *tmp_decl=
      (SgClassDeclaration *)(isSgClassType(tmp_n)->get_declaration());

  // Lookup the declaration of the value member function
  // associated with the declaration of sync classes.
  SgMemberFunctionSymbol *fsym=0;
  SgName nm=TRAN_STRING::value;
  fsym=(SgMemberFunctionSymbol *)
            (tmp_decl->get_definition()->lookup_function_symbol(nm));

  // Get line number information for variable.
  Sg_File_Info *finfo=e->get_file_info();

  // Create a value function call
  SgMemberFunctionRefExp *fref=new SgMemberFunctionRefExp(finfo,fsym);

  // Create the actual member function call: n.value()
  SgDotExp *dotexpr=new SgDotExp(finfo,e,fref);
  SgFunctionCallExp *fcall=new SgFunctionCallExp(finfo,dotexpr);

  // Convert function call to a generic expression
  SgCastExp *fcast=new SgCastExp(finfo,fcall,t,0);

  return fcast;
```

Program 5.1: Sage 2.0 Transformation for Sync Variables

5.3 Source to Source Transformations

In this section, we provide a brief overview of the transformation process used by the CC++ compiler to convert a CC++ program to a form that is acceptable to an unmodified C++ compiler. The result of the transformation process

is a ANSI C++ program, with references to a CC++ class library, which provides base classes that implement basic functionality for `sync` variables, `global` pointers and `atomic` functions. The class library is in turn built on top of an underlying runtime system. We have used the Globus Grid Toolkit [FK97] as our runtime environment.

The transformations are decomposed into two passes over the Sage 2.0 program structure. During the first pass, `global` classes are recorded and their type converted to a regular C++ class. In addition, variable declarations with the `sync` modifier are converted to a declarations of a corresponding Sync class that supports the synchronization operations. We then examine all expressions in the file, turning read operations on `sync` variables into explicit calls to the `value` member function, as described above.

The second pass performs a range of transformations on classes, statements, declarations and expressions. The following is an incomplete list of some of the transformations performed during the second transformation pass by the compiler:

- Class based transformations. If the function has any atomic functions, a special atomic virtual base class is added. This class ensures that one lock is shared across all atomic functions called on an instance of the class. Access to a class via a global pointer is supported by the addition of a number of *entry* functions which provides an interface to a remote procedure call provided by the underlying runtime library. One entry function is added for each member function of the class as well as an entry function to enable remote reading and writing of data members. The compiler also adds additional member functions to facilitate access to private members from a parallel block.

- Declaration based transformations. All global pointer declarations are converted to be a single generic global pointer type. The `atomic` keyword is stripped from function declarations and a variable whose constructor calls a lock function in the atomic base class is added to the beginning of `atomic` function definitions.

- Statement based transformations. Library based thread creation functions typically require a pointer to a regular C function. The CC++ compiler must extract each statement from a parallel block or loop and encapsulate it as a static member function of a class, or a global function. In addition, detection of termination of parallel blocks must be added using a barrier defined in the CC++ runtime library.

- Expression based transformations. The compiler transforms expressions involving global pointers into a sequence of buffer packing operations (e.g. argument marshalling) and remote procedure calls. The compiler generates specialized protocols depending on if a function call is synchronous,

or asynchronous and if the call has a return value. Expressions causing processor object creation and deletion (i.e. `new` and `delete` operators of `global` classes) are converted into a sequence of resource management operations that create a process running the executable for the processor object on a remote computing resource. In addition, processor object references (i.e. `::this` expressions) are removed and replaced with references to a static variable which contains a pointer to a local instance of the current processor object.

5.4 The Runtime Environment for CC++

Recent trends in high-performance networking are resulting in an significantly increased availability of high-bandwidth network connections. With these advances in networking infrastructure, it is now possible to construct large-scale distributed computing environments, or "computational grids" as they are sometimes termed [FK98]. Computational grids provide an application with predictable, consistent and uniform access to a wide range of remote resources, including compute resources, data-repositories, scientific instruments, and advanced display devices. This access makes it possible to construct whole new classes of applications, such as supercomputer enhanced instruments, desktop supercomputing, tele-immersive environments, and distributed supercomputing [CS92].

In a separate DARPA funded project, we have been investigating the concept of computational grids, and have developed a runtime infrastructure, called Globus, to support development of computational grid applications. Globus consists of a toolkit which provides basic Grid services, including:

- mechanisms for thread creation,
- thread synchronization primitives,
- communication mechanisms, and
- security
- resource management, i.e. remote process creation and control.

We have constructed the CC++ compiler to target the Globus grid toolkit for runtime support. In doing so, we enable CC++ programs to execute not only on parallel computers, but in distributed Grid environments as well.

One of the more interesting uses of Globus in the CC++ runtime library it the use of the Globus Resource Allocation Manager [CFK+98] (GRAM), to support processor object creation. GRAM provides a uniform interface to a wide range of local resource management services, such as LoadLeveler, PBS, NQE, and LSF. The GRAM API's take a standardized resource description, authenticate to the remote resource, allocate processors on that resource, and

initiate the execution of the specified program on those processors. In addition, GRAM supports automatic remote staging of the application binary on the remote resource.

The CC++ compiler generates a separate executable for each processor object and uses calls to GRAM to start the execution of the processor object on a remote resource. As described above, a `proc_t` object is provided as an argument to a `new` of a `global` class to specify what computer should be used to execute the processor object. The CC++ compiler converts the information in the `proc_t` structure to a Globus resource specification and then issues a GRAM call to allocate the requested resources. Because GRAM handles all local resource management and security issues, the amount of code that must be generated by the compiler is minimized.

6 Application Demonstration

To illustrate the power of CC++ as a Grid programming language, we developed a application of interest to DoD, in collaboration with colleges at the Aerospace Corporation. This application, called NEPH, enables near real-time detection of clouds from satellite imagery.

The purpose of this application is to process two-dimensional infrared and visible light images from on-board satellite sensors, such as DMSP, and locate the position and elevation of clouds from these images. The steps in this processing include:

- geo-locate the images by assigning a latitude and longitude value to each pixel,
- correct infrared data for previously known temperature climatology depending on geography type (land, water, desert, etc.), time of day and time of year,
- correct Visible light data for previously known background brightness according to geography type, time of day and time of year,
- Use the corrected data and other historical data is used to determine thresholds in visible and infrared channels to decide if a pixel is clear, partially cloudy or completely cloudy
- Apply the thresholds to the images to produce a cloud map. Use the infrared values to produce a elevation map. Combine these two maps to produce a three dimensional representation of cloud location.

Figure 2 shows a three dimensional visualization of the output produced by this application. Red indicates cloudy pixels detected only in the infrared image. Green indicates cloudy pixels detected only in the visible light image. White and gray indicate agreement.

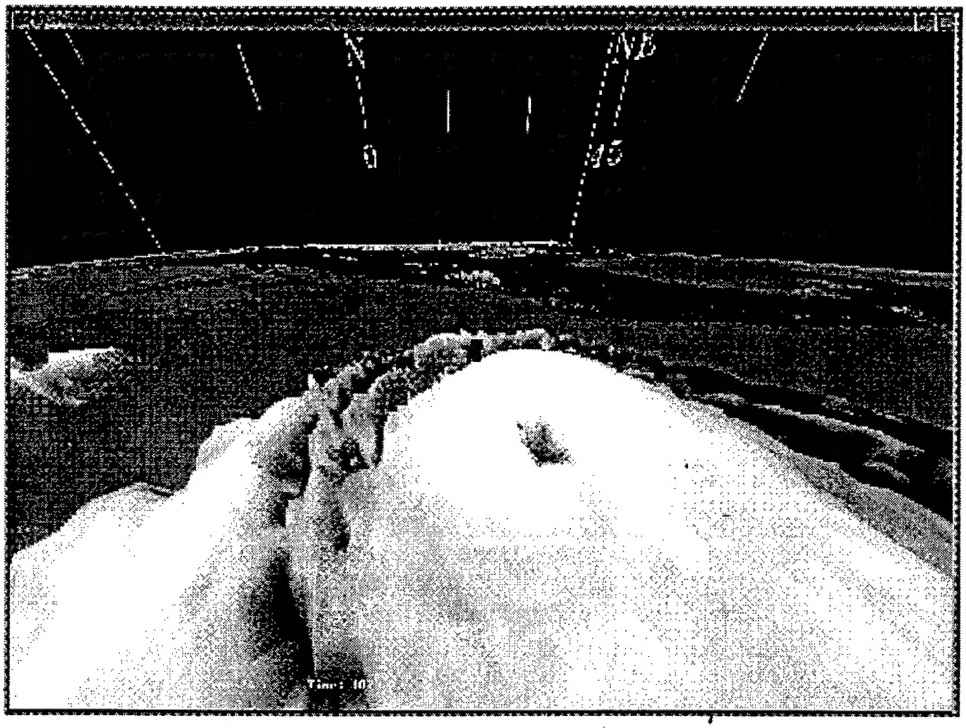

Figure 2: Stereoscopic view over top of hurricane towards Baja California and Mexico. Vertical scale is exaggerated.

To simplify the design of this application, it was coded in CC++. The application consists of three different types of processor objects: an input object, a cloud detector object and a visualization object. The cloud data is stored on a workstation machine which has direct access to the downlink data from the satellite. The cloud detection process is compute intensive, and is best done on a parallel machine. Finally, the visualization component needs to be located on the machine that the end user is seated at, which is more then likely not the parallel machine. Because of its need for distributed heterogeneous resources, NEPH is an example of a Grid application.

By coding NEPH in CC++ we were able to dramatically simplify the coding process. The program is started from the visualization console. The program creates processor objects on the machine on which the the image data is located, and a collection of cloud detector processor objects on a parallel machine. Communication between processor objects is performed via global pointer operations.

The multithreading capabilities of CC++ are used to simultaneously integrate processed data from a cloud detector and visualize the results.

Because CC++ is built on the widely deployed Globus infrastructure, the user of NEPH authenticates once to the Grid environment and then Globus provides strong authentication to all of the resources used by the application. Globus hides the complexity of the fact that different communication protocols may be used between nodes in a parallel machine and between separate computers, thus simplifying the task of configuration. Finally, because Globus provides a uniform interface to the different local resource management tools that may be running on a parallel platform, the application as a great deal of flexibility in the selection of resources used to run the cloud detectors.

7 Summary

In this report, we summarized the results of our efforts in developing a unified approach to supporting parallel programming in C++. The central accomplishment of this project has been to develop a robust implementation of the CC++ language. As part of this process we have also developed basic infrastructure that can be used for developing a range of programming tools for C++. The utility of the CC++ compiler has been demonstrated on a number of applications. As part of this project, we have developed a general tool for source to source transformations on C++ programs. This tool has been used for a range of applications, including CORBA IDL compilers and performance measurement and visualization tools.

Our CC++ compiler generates calls to the Globus Grid toolkit. By choosing this as are target execution environment, we have been able to track the latest developments in Computational Grid environments. Consequently, we have demonstrated the use of our CC++ compiler on high-performance distributed applications as well as more traditional parallel applications.

References

[Ack82] William B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, feb 1982.

[Bin93] Ulla Binau. Distributed diners: From unity specification to cc++ implementation. Technical Report CS-TR-93-20, California Institute of Technology, 1993.

[BOCF92] A. P. W. Böhm, Rodney R. Oldehoeft, David C. Cann, and John T. Feo. SISAL reference manual, language version 2.0. Technical report, LLNL, 1992.

[CFK+98] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[CK92] K. Mani Chandy and Carl Kesselman. The derivation of compositional programs. In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming.* MIT Press, 1992.

[CS92] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[CT91] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Bartlett and Jones, 1991.

[ES90] Margret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[FBACX94] I. Foster, B. B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates high performance fortran and fortran m. In *Proc. 1994 Scalable High-Performance Computing Conf.* IEEE Press, 1994.

[FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[FK98] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann Publishers, 1998.

[Gri93] Andrew S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[Siv94] Paolo Sivilotti. A verified integration of parallel programming paradigms in CC++. In *Proceeding of the International Parallel Processing Symposium*, 1994.